- Data words are listed with the location counter value first, followed by the data values themselves. If the word has been divided into multiple parts, MASM shows those parts separated by one or more blank columns.

- Instruction words are also listed with the location counter value first. But they are formatted with the f, j, a, x, h and i together, and u fields separated by spaces.

At each location counter change and at each page eject, MASM puts the location counter number to the left of the location counter value.

After the assembly listing itself, MASM prints:

- The final values of the location counters

- The list of external references (undefined symbols)

- The list of entry points (externally defined symbols)

- The cross-reference listing, if requested.

MASM may also produce other outputs, depending on its release level.

The last line produced by MASM is a summary line. If there are any assembly error flags (see next section), the types and quantities will be noted here.

5.4 Assembly Error Flags

In some ways, MASM is a rudimentary piece of software. It does not create a fancy English error message when it detects a syntax or other error. Instead, the offending line is flagged with one or more letters beginning in column 1. It is up to you to find the exact problem MASM detected. Here are some of the flags you may encounter:

- D - duplicate definition of an implicitly defined label. You probably used the same symbol in the label field of one or more lines.

- E - an error in syntax. This could be a problem almost anywhere in the flagged line.

- I - instruction mnemonic not recognized. You have misspelled an instruction mnemonic or procedure name. Or, perhaps you misspelled the label on a procedure definition (MASM generates a NOP—no operation—instruction.)

- Q - quote terminator missing on a string. For example:

```
MSG050      'TAPE LABEL MISSING  .
```

- T - truncation error. You tried to cram a value into a field which was too small for it. For example, label MSG050 is at 0150 relative to location counter 2, and you have coded:

```
            L           MSG050  .
```

Obviously, the line is missing the register number, but MASM tried to cram 0150 into the a field of an instruction word, resulting in a truncation error. This line would probably also be flagged with an E flag.

- L - level error. This is usually caused by omitting the $END directive at the end of a procedure definition or at the end of the source.

- R - relocation error. This is caused by giving MASM a construct which cannot be placed accurately in the relocatable output element, such as $/2.

You might want to pull out your MASM reference at this point, and familiarize yourself with the location of the manual's definitions of error flags.

Chapter 6: Basic Integer Operations

6.1 Moving Data In and Out Of Registers

Sixty percent of MASM programming uses simple operations:

- Copying file names, counters and such from one place to another.
- Keeping simple tallies, counting objects, and so forth.

These operations can be done with the instructions described in this chapter. (Hint: You need to understand everything in this and the next chapter before you go on.) It will help you to keep either a Quick-Reference Guide or a Processor and Storage Reference close by as you read this chapter. (Any 1100/2200 Processor and Storage Reference will do.)

While you can do a few simple arithmetic operations without using a register, you can do the interesting ones only with a register. There are three instructions which can load a register. The first is LA (Load Accumulator). (In this and other examples, data and instruction words are shown in consecutive cells. In practice, they would be separated into even and odd location counters. The location counter numbers do not affect the operation of the instructions.) Here is an example of LA:

```
COUNT1      +           5
            LA          A0,COUNT1  .
```

This instruction copies the contents of the cell COUNT1 into accumulator A0. COUNT1 is not altered. Since there are 16 A registers, you could just as easily code:

```
COUNT1      +           5
            LA          A15,COUNT1  .
```

as long as you wanted the contents of COUNT1 to go into the last A register.

Next, suppose that you needed to copy the contents of COUNT1 to another cell, called TALLY1. This is one way to do it:

```
COUNT1      +           5  .
TALLY1      +           0  .
            LA          A5,COUNT1  .
            SA          A5,TALLY1  .
```

This example introduces the next instruction, SA (Store Accumulator). SA copies the contents of the named A register to the destination address. In this case, the destination is TALLY1.

The CPU performs actions requested by instructions, not MASM source. Therefore, at the outset, consider the instruction words created from your source lines. For the LA instruction, here are the fields of the instruction word:

```
  f     j     a     x     h     i     u
-------------------------------------------
010     00    05    00     0     0   02000    (LA A5,COUNT1)
 01     00    05    00     0     0   02010    (SA A5,TALLY1)
```

This example assumes that COUNT1 was assembled at location 02000 of its location counter, and TALLY1 at 02010.

As you encounter each new instruction, locate it in one of the instruction summary tables. These tables are found in the Operations and Programming Quick Reference Guide and each Processor and Storage Reference. (If there are two tables, refer to the one for basic mode only.) They condense the operation of each instruction to just a few lines, using a notation that may seem cryptic at times, but is quite concise. For example, the description of SA (f = 01, entries in order by ascending f) shows this entry:

(A_a) -> U

This entry means that the CPU copies the contents of the A register specified by the a field (A_a) to the destination (U). Can you find the entry for LA (f value 010)?

You can also move data in and out of X or R registers, using these instructions:

- LX - Load X
- LR - Load R
- SX - Store X
- SR - Store R

Locate the summary description of LX, Load X, f=027. It is:

(U) -> X_a

Don't let the subscript "a" fool you. This instruction loads the X register the number of which is given by the a field of the instruction. (The x field is used for something else, described later.) This is what an LX instruction looks like in source code:

```
INDEX1      +           15  .
            LX          X5,INDEX1  .
```

For most instructions, the a field names a register out of a group. Which group is used by the CPU depends on the op code, f. This instruction, even if not rejected by MASM, would not give the desired results:

```
INDEX2      +           12  .
            LX          A5,INDEX2  .
```

The instruction mnemonic LX wants an X register, and you have given it an A register. On the other hand, this line works:

```
            LX          A3,INDEX2  .
```

because A3 is an index register.

LR loads R_a from the operand address:

```
        LR          R7,COUNT3   .
```

If you want to load a given register, MASM can accept a generic mnemonic with an X, an A, or an R register:

```
WORD01  +           100 .
        L           A5,WORD01   .
```

The mnemonic works on any of the three types of registers. The actual op code generated from the mnemonic depends on the type of register.

- If the register is an A register, L becomes the op code for LA.
- If the register is an X register, L becomes the op code for LX.
- If the register is an R register, L becomes the op code for LR.

There is also a generic mnemonic for SX, SA, and SR: S. This mnemonic is assembled following the same rules as for L.

6.2 Changing The Sign

There are operations which can change the sign of data as it is copied in or out of a register. These operations work only on A registers.

The first pair "flip", or reverse, the sign of data as it is transferred:

- LN or LNA (Load Negative A) reverses the sign of a quantity as it is loaded.
- SN or SNA (Store Negative A) reverses the sign as it stores data.

(Each of these instructions has two mnemonics. The operation of LN and LNA is the same whichever mnemonic you use; ditto for SN and SNA.) Here is an example of LNA and SNA:

```
TALLY1  +           12  .
COUNT1  +           5   .
        LNA         A5,TALLY1   .
        SNA         A5,COUNT1   .
```

The LN performs these steps:

- Copies the contents of TALLY1 to the arithmetic section of the CPU. (Thus, TALLY1 is not changed).
- Makes the quantity negative by reversing all of its bits.
- Copies the quantity to A5.

The result in A5 is -12 (octal 0777777777763). Remember, 12 is octal 014; reverse all the bits, and the result is 0777777777763. (Note that the leading zero is not part of the value, it is just the octal indicator.)

SN performs the same operations as LN, only in reverse order, that is, storing a register. The example just shown will put +12 (000000000014) in COUNT1, replacing the 5 originally there.

Another pair of instructions changes the sign conditionally. The LMA (Load Magnitude A) and SMA (Store Magnitude A) instructions transfer the absolute value of the operand to the destination. If the sign of the operand is clear (a positive quantity), the operand is not changed. If the sign is set (a negative quantity), the bits of the operand are reversed, making it positive.

Here is an example using SMA:

```
TRACK1      +           12  .
WORK01      +           0  .
            LNA         A0,TRACK1  .
            SMA         A0,WORK01  .
```

The result in WORK01 is +12 (014). The LN places a -12 in A0, and the SMA transfers the absolute value of -12, that is, 12, to WORK01.

One final load instruction in this group has no counterpart among the store instructions. LNMA (Load Negative Magnitude A) takes the absolute value of the operand, reverses each bit to create a negative quantity and places that in the named A register. Here is an example of the LNMA instruction:

```
COUNT1      -           5  .
            LNMA        A15,COUNT1  .
```

The result in A15 is -5. The magnitude (absolute value) of -5 is 5, and the negative of 5, that is, -5, is transferred to A15.

6.3 Basic Arithmetic

Once your program has loaded a number into a register, it can begin to add to it, subtract from it, multiply it by other quantities, and so forth.

The simplest operation is AA (Add to A). It adds the operand to the contents of the named A register. This example adds the contents of TRACK1 and TRACK2, placing the result in ADRS03:

```
TRACK1      +           12  .
TRACK2      +           503  .
ADRS03      +           0  .
            L           A3,TRACK1  .
            AA          A3,TRACK2  .
            S           A3,ADRS03  .
```

The result in ADRS03 will be 515. By the way, the CPU will not forget that result after this

computation is finished. Hundreds, thousands, even millions of instructions later, if the program has not altered A3, that 515 will still be there.

Subtraction is another basic arithmetic operation. The 1100 mnemonic for subtraction is ANA (Add Negative to A). This example subtracts TRACK2 from ADRS03 to get TRACK1:

```
TRACK1      +          0  .
TRACK2      +          12 .
ADRS03      +          503 .
            L          A1,ADRS03 .
            ANA        A1,TRACK2 .
            S          A1,TRACK1 .
```

Your program can do this same sort of simple arithmetic with X registers, too. This example combines the AX (Add to X) and ANX (Add Negative to X) to implement a simple formula (CELL07 = LOCATN + OFFSET - BACKWD):

```
LOCATN      +          040110 .
OFFSET      +          12 .
BACKWD      +          2 .
CELL07      +          0
            L          X11,LOCATN .
            AX         X11,OFFSET .
            ANX        X11,BACKWD
            S          X11,CELL07 .
```

You may want to calculate the result in X11 for yourself.

The steps in rendering a formula into code are similar to the steps used to evaluate the formula by hand. If there are only addition and subtraction in the formula, perform the steps of the formula from left to right. If multiplications or divisions are present, they must be performed before the additions and subtractions. If there are parentheses in the formula, the quantities inside the parentheses must be computed first.

Like the load instructions for X, A and R registers, the add and add negative instructions for A and X registers have generic forms. MASM will assemble an A mnemonic as an AA op code if the line names an A register, and as an AX op code if the line names an X register. Similarly, AN will be assembled as ANA if the line names an A register, and as ANX if the line names an X register.

Here is a recap of the generic mnemonics described so far:

Specific Mnemonics	Generic Forms
LX, LA, LR	L
SX, SA, SR	S
AA, AX	A
ANA, ANX	AN

Using the generic or specific mnemonics is your choice.

If you need to add to the contents of a register, but preserve the register, use the AU (Add Upper) instruction. This instruction performs the following steps:

- Move the named register's contents to a work register in the CPU's arithmetic section.
- Add the operand to the work register.
- Transfer the result to the A register numbered one higher than the named register (that is, A_{a+1}).

Thus, the contents of A_a are not disturbed. Simpler calculations (C = A + B - D) don't need the AU instruction. You will need it more often when one sequence of instructions must compute several things at once.

There is also an ANU (Add Negative Upper) instruction. It works just like the AU instruction, except that it subtracts from the work register. As before, the result goes to register A_{a+1}. A_a is not affected.

6.3.1 Increment and Decrement Instructions

Suppose your program needs to bump a number of counters. (Definition: Bump means "add 1 to".) Done as ordinary arithmetic, you might code this sequence:

```
TRACK     +          0  .
CONST1    +          1  .
          L          A0,TRACK  .
          A          A0,CONST1  .
          S          A0,TRACK  .
```

(Actually, there is a simpler way to add a constant to a register, but that's getting ahead of things.) The method shown above takes 3 instructions and 3 memory accesses. The INC (Increment) instruction does the same job, and would make the example look like this:

```
TRACK     +          0          .
          INC        TRACK      .
          NOP                   .
```

(The NOP instruction does "no operation", that is, nothing but take up an instruction word. It is not part of the INC instruction.) The INC adds 1 to the operand, and it needs only two memory accesses.

The INC instruction has another function. If the initial value of the operand is zero, or the value after adding 1 is zero, it executes the next instruction (abbreviated NI). Otherwise, it skips NI. (If you don't care to use the test feature of INC, just follow it with a NOP as shown.) You can use INC in combination with a jump and a carefully chosen value to execute a loop a certain number of times. If that seems too complicated, don't worry. Stick with using INC to bump counters, and don't forget to put an NOP after each one. (A later chapter will deal with loop controls.)

Likewise, there is a DEC (Decrement) instruction. Similar to INC, it subtracts 1 from the operand. There are also INC2 (Increase Operand by Two) and DEC2 (Decrease Operand by Two) instructions. All three of these test the operand and result for zero. If nonzero, NI is skipped.

INC, DEC, INC2, and DEC2 do not need registers. Thus, they:

- Increase the number of loop counters your code can maintain at once.
- Simplify code which requires all available registers for other purposes.
- Reduce the number of machine cycles needed for the very simple arithmetic they can handle.

Finally, INC and DEC do their functions under what is called "storage lock". That means that if two or more activities try to update the same counter in memory at one time, each will lock the counter in turn and make its change. The storage lock feature will be important for multi-activity programs (beyond the scope of this text).

6.4 Instructions For Storing Constants

Certain constants show up so frequently, a set of instructions were implemented to store them. These constants and corresponding instructions are:

Constant	Mnemonic	Title
Binary Zero (000000000000)	SZ	Store Zero
Negative Zero (0777777777777)	SNZ	Store Negative Zero
Binary One (000000000001)	SP1	Store Positive One
Negative One (0777777777776)	SN1	Store Negative One
Fieldata Spaces (050505050505)	SFS	Store Fieldata Spaces
ASCII Spaces (040040040040)	SAS	Store ASCII Spaces
Fieldata Zeros (0606060606060)	SFZ	Store Fieldata Zeros
ASCII Zeros (060060060060)	SAZ	Store ASCII Zeros

These instructions save the memory space and overhead of loading the constant into a register, then storing it.

These instructions, since they do not involve any register, use the a field of the instruction word as an extension of the function code (f) field. (Please take a moment and look them up in your instruction summary table. Note the a field values used by the instructions.)

You can use each of these instructions with a partial word modifier (see Chapter 7) to change just a part of a word. Note that the value stored by the instruction is right-justified in the named partial word. For example, a Store Negative 1 into a sixth word stores 076, not 077.

6.5 Multiplying Your Options: More Arithmetic Instructions

Rounding out this discussion of integer arithmetic are the operations of multiplication and division.

Two of these instructions differ from those described so far; they may involve more than one register. The reason for using more than one register is easy to see. Multiply 1000 by 1000. The result, 1,000,000, requires 3 more digits than either of the factors. In computer multiplication, the product of two 36-bit factors could extend to 72 bits. In MASM programming, though, few numbers get that large.

The MI (Multiply Integer) instruction multiplies the contents of A_a by the operand. The result is placed in A_a and A_{a+1}. The most significant (left) 36 bits end up in A_a, the least significant (right) in A_{a+1}. This example multiplies two numbers using MI:

```
TRACK       +           28  .
SECTOR      +           0   .
CONST64     +           64  .
            L           A0,TRACK  .
            MI          A0,CONST64  .
            S           A1,SECTOR  .
```

This code fragment multiplies TRACK by 64 to get SECTOR. The result, 1792, is stored from A1, not A0. Upon completion of the MI instruction, A0 contains 0, the most significant half of the result; A1 contains 1792, the least significant. Unless the significant bits of the factors total at least 36 (divided 18 in A_a and 18 in (U), or 16 in A_a and 20 in (U), or other combination), the result in A_a will be 0.

If you know the result will never be more than 36 bits (and this is true the vast majority of the time), you can use the MSI (Multiply Single Integer) instruction. It multiplies A_a by the operand and places the result back in A_a, thus requiring only one register. Let's repeat the previous example, using MSI:

```
TRACK       +           28  .
SECTOR      +           0   .
CONST64     +           64  .
            L           A0,TRACK  .
            MSI         A0,CONST64  .
            S           A0,SECTOR  .
```

The product ends up in A0 this time. A later chapter will disclose ways to make multiplying by constants and powers of 2 easier than shown by this example.

The DI (Divide Integer) instruction's result, like MI's, also requires 2 registers. In fact, DI is almost exactly the reverse of MI. DI requires a 2-register dividend (the quantity being divided) in A_a and A_{a+1}. There are two results: a quotient in A_a and a remainder in $A_{a+}1$.

The quotient in integer division represents the whole number of times the divisor must be added to itself to obtain the dividend. That is, the quotient times the divisor will always be less than or equal to the dividend. For example, 13 divided by 5 equals 2 with a remainder of 3.

It is easy to forget to expand a 1-word operand into 2 registers before doing a DI instruction, giving unpredictable results. Don't let it throw you.

Here is an example using DI:

```
WORDS      +          2000 .
SECTORS    +          0          .
OFFSET     +          0          .
CONST28    +          28         .
CONST0     +          0          .
           L          A13,WORDS  .
           L          A12,CONST0 .
           DI         A12,CONST28 .
           S          A12,SECTOR .
           S          A13,OFFSET .
```

The aim of this code fragment is to take the contents of WORDS (2000) and create a sector address and an offset within that sector. (You may need to do this calculation for disk input/output.) The sector address is WORDS/28 (integer division), and the offset within that sector is the remainder.

In this example, A_a is A12; A_{a+1} is A13. The first instruction moves the non-zero part of the dividend to A13. The second instruction zeros out the most significant (left) part of the dividend, A12. (A simpler way to do this step will be introduced later.) The DI instruction does the division itself. The fourth instruction saves the quotient, and the last instruction saves the remainder.

To summarize these last three instructions:

- DI accepts a two-word quantity and creates a one-word quotient and a one-word remainder.
- MI accepts 2 one-word operands and creates a two-word product.
- MSI accepts 2 one-word operands and creates a one-word product.

6.6 X and A Registers Overlap: The Plot Thickens

The discussion of user registers mentioned that four X registers double as A registers. The registers named as X registers are numbered X1 through X11. The A registers are numbered A0 through A15. The overlapping registers are A0 (X12) through A3 (X15). Registers A0 through A3 can serve as index or accumulator (arithmetic) registers.

While this overlap might seem complicated at first, in fact the overlap simplifies coding. Calculations of index values can involve multiplying an entry size by an entry number; the multiplication requires an A register. Addressing the table entry needs an X register. The calculation and indexing can be done in the same register.

Note that MASM generates the X-type op codes for these instructions when used with the overlapping registers. For example, if you code "L A0,LABEL", MASM generates the op code for LX. (The a field will be 12, for the twelfth X register, A0.) Next time you have an assembly listing handy, check the object code column. Look at the op codes assembled for generic mnemonics.

6.7 Double-Word Instructions

Another group of instructions performs fixed-point arithmetic and logical operations on 72-bit operands, two adjacent A registers at once. The first group of double-word instructions perform arithmetic just as did the single-word instructions.

Not counting the (possibly) two-sign bits, the remaining part of the operand is still twice as many bits wide (70 bits versus 35 bits), allowing your program to handle much larger numbers. The largest signed number you can fit in 36 bits is very large, but some counters in the 1100 count very small units in very large quantities, such as how many CPU units of 60 nanoseconds your program has consumed. So, 72 bits is not at all unreasonable.

These instructions operate on single, 72-bit operands, not pairs of single-word operands. (Note that the 1100 does not place restrictions on which registers may be used for double-word instructions, as some other systems do.)

Here are the instructions:

- DL - Double Load A
- DS - Double Store A
- DA - Double-Precision Fixed-Point Add
- DAN - Double-Precision Fixed-Point Add Negative
- DLN - Double Load Negative A
- DLM - Double Load Magnitude A

The first pair is Double Load (DL) and Double Store (DS). These work just as their single-word counterparts, Load A and Store A. Like LA and SA, DL and DS work only on A registers:

- DL loads the contents of U and U+1 into A_a and A_{a+1}, respectively.
- DS stores the contents of A_a and A_{a+1} in U and U+1, respectively.

Note that only a single value is defined for the j field of each of these double-word instructions. This means that you cannot specify a partial-word modifier (like H2) with any of them. The result will probably be a T (truncation) or E (error) flag from MASM; your program will not do what you wanted. A DL or DS will always load or store 2 complete words.

The next pair, Double Add and Double Add Negative, also work just as their 36-bit counterparts, Add to A and Add Negative to A:

- Double Add performs a 72-bit signed addition. It adds the contents of U and U+1 to the contents of A_a and A_{a+1}, and places the result in A_a and A_{a+1}.
- Double Add Negative subtracts the contents of U and U+1 from A_a and A_{a+1}, and places the result in A_a and A_{a+1}.

The last pair, Double Load Negative A and Double Load Magnitude A, work the same as Load Negative A (LNA) and Load Magnitude A (LMA), respectively, only on two-word operands.

- Double Load Negative A obtains the contents of U and U+1, creates the negative of it by flipping each bit and places the result in A_a and A_{a+1}.

- Double Load Magnitude A obtains the contents of U and U+1; if the sign bit is positive, the value is loaded into A_a and A_{a+1}. If the sign bit of U and U+1 is negative, each bit of the value (including the sign) is flipped, and the result is loaded into A_a and A_{a+1}. (Recall that "magnitude" in this context means the algebraic absolute value of U and U+1, |U,U+1|.)

Here is an example of double-word fixed-point arithmetic. It uses small values to keep it simple:

```
OPND1     +        27D        . (OCTAL 033)
OPND2     +        122D       . (OCTAL 0172)
RESULT    $RES     2          .
          DL       A5,OPND1   .
          DAN      A5,OPND2   .
          DS       A5,RESULT  .
```

Can you figure out the octal value in RESULT? Here are the values in A5 and A6 at each step:

```
                  A5              A6
1. DL:      000000000000    000000000033
2. DAN:     777777777777    777777777640
```

One easy way to do the subtraction, since we know the result is negative, is to subtract 033 from 0172, and take its negative by flipping all the bits:

```
     A5              A6
000000000000    000000000172
000000000000    000000000033
------------    ------------
000000000000    000000000137 and flip all the bits,
777777777777    777777777640
```

The sign bit of A6 will be propagated into A5.

Exercises

Write MASM code needed to do the following things:

E6-1. Load A2 from TRACK1.

E6-2. Store A7 to SECT27.

E6-3. Copy the contents of SUB1 to SUB2.

E6-4. Copy SUB1 minus 5 to SUB2.

E6-5. Calculate SECT01 = WRDOFF / 28. (Discard the remainder.)

E6-6. Subtract 1 from WDGCNT without using a register.

E6-7. Calculate VAR015 = WRDOFF / 28 + WRDOFF + 100. Load from WRDOFF only once.

(Hint: don't change WRDOFF in the register it is first loaded into. There is more than one way to solve this problem, but you will have to use more than one register.)

E6-8. Blank out FILNAM (2 words long) with Fieldata spaces.

E6-9. Repeat E6-1 through E6-4 using double-word fields and instructions to match.

Chapter 7: Register Addresses and Partial Words

7.1 Registers Have Addresses, Too

The registers (X1-X11, A0-A15, and R1-R15) do not exist in an addressing vacuum. They are assigned addresses, similarly to memory locations:

- X1 through X11 are at addresses 01 through 013.
- A0 through A15 are at addresses 014 through 033.
- R1 through R15 are at addresses 0101 through 0117.

The other addresses below 0200 are divided between:

- Exec-only X, A and R registers, and
- Locations defined by hardware functions such as addressing and interrupts.

Your program cannot access registers other than the user X, A and R registers.

All the registers are below the lowest program-relative address, 01000. Therefore, no ambiguity can arise between a register address and a user program address.

Don't worry about having to memorize register addresses. What these addresses mean is that you can code a register in the u field. For example:

```
L       A0,X11  .
S       A1,R2   .
```

You will need to move information between registers in this fashion in complicated formulas, or code which computes several quantities at once. Or, in some cases, you might have to use an A register to calculate a loop counter which will reside in an R register. (The R registers cannot perform ordinary, direct arithmetic, such as add, subtract, multiply, and so forth. Instructions for loop control can increment and decrement R registers, however.)

7.2 Partial Words: Carving Up Memory Locations

The 1100 CPU can address parts of words. It can divide a 36-bit word 5 ways:

- The whole word, W.
- Halves, H1 and H2. Each half word is 18 bits wide.
- Halves, XH1 and XH2. Each half word is 18 bits wide.
- Thirds, T1-T3. Each third word is 12 bits wide.

- Quarters, Q1-Q4. Each quarter word is 9 bits wide. Due to the hardware design, it is inconvenient to use third words and quarter words in the same program. More on this later.

- Sixths, S1-S6. Each sixth word is 6 bits wide.

Figure 7-1 shows these word divisions graphically. These divisions, called partial words, never span words. If your program must access fields which start or end on other than these boundaries, it must perform the necessary shifts itself.

The names shown in the list above (H1, T2, and so forth) are mnemonic names for various values of the j field of an instruction word. These names are variously known as partial-word modifiers or partial-word designators.

In your Processor and Storage Reference or Quick-Reference Guide, locate a table called "j-Determined Partial Word Transfers". (In the Quick-Reference Guide, UP-7824.3, there is one on pages 5-20 and 5-21.) You need to become familiar with this table and its contents. For each value of j, the table tells which partial word is transferred from memory to CPU and vice versa.

In the Quick-Reference Guide, this table is followed by a table titled "GRS Control Register Values and j-Values for Addressing Partial Words". Put simply, this table shows what is in AXR$: definitions of register names and partial-word modifiers. For example, R1 is 0101, A15 is 033, and S1 (a partial word) is 015. Use this table along with the previous table to check the exact bits of a word transferred by a given partial word-modifier.

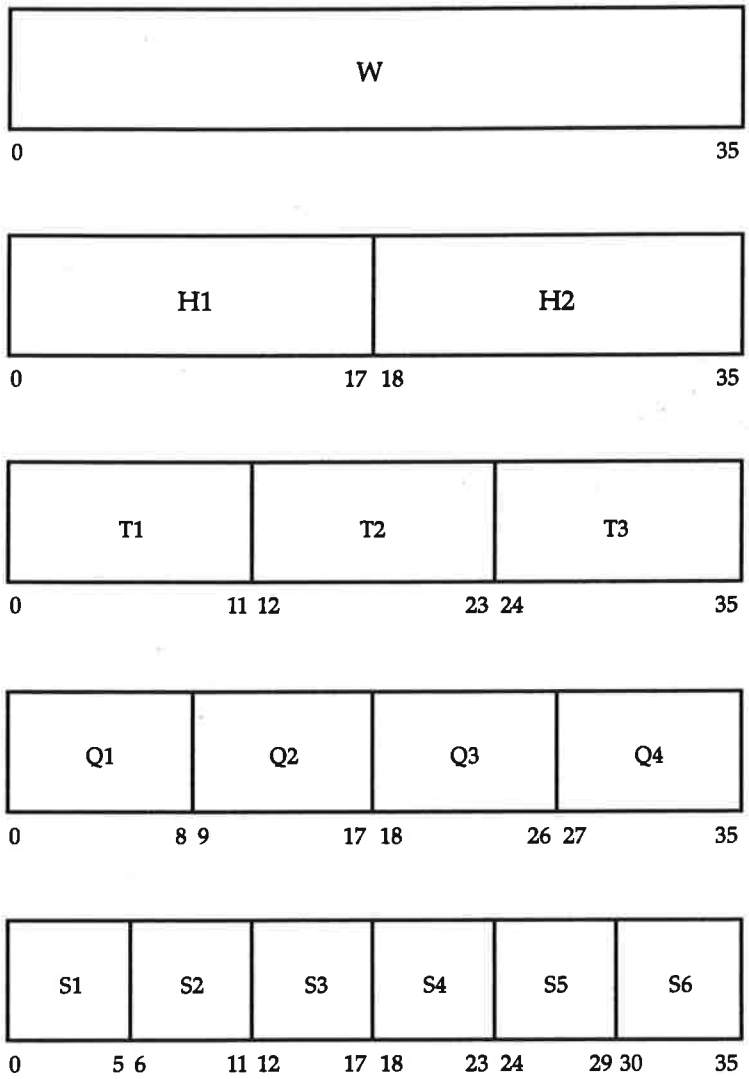

Figure 7-1. Partial Words

Note that when you code a partial word-modifier on an instruction, such as L,H2, MASM stores the value 1 (the definition of H2 in AXR$) into the j field of the instruction word. The CPU interprets the 1 to transfer the rightmost 18 bits of an operand into its arithmetic section.

Below are the actions taken for some j values. As you read these descriptions, please follow them in your summary tables. (Since the bits are numbered differently on different machines, note the placement of the bits, rather than their numbers.) Some of the j values are omitted in this list, for reasons to be explained.

- W (j=0) - The entire operand word, 36 bits, is transferred.
- H2 (j=1) - The right half of the word, 18 bits, is transferred.
- H1 (j=2) - The left half of the word (again, 18 bits) is transferred. (Note that the j value for H2 is 1, and for H1 it is 2, the opposite.)
- XH2 (j=3) - The right half of the word is transferred with sign extension. This means that the leftmost bit of H2 is treated as a sign; the left half of the register is filled with that sign.
- S1 (j=015) - The leftmost 6 bits of the operand are transferred.
- S2, S3, S4, S5, S6 (j=014, 013, 012, 011, 010) - Six-bit parts of the operand are transferred. S6 is the rightmost 6 bits.

These descriptions say x number of bits "are transferred". There are two directions in which transfer is possible, from memory to CPU, and from CPU to memory. In those two directions, here is what is meant by "are transferred":

- Memory to CPU - The CPU acquires the entire operand word, discards all but the desired bits, right justifies the field as necessary and moves it into the named register. The remainder of the register is zero-filled or sign-filled, according to the j value.

In other words, for H1 and H2, the remaining 18 bits on the left are zero filled; for S1 through S6, the remaining 30 bits on the left are zero filled. For XH2, the left half of the register is filled from the sign (leftmost) bit of H2. If that bit is zero, the left half of the register is zero; if set, the left half of the register will be 07777777, all 1 bits.

- CPU to Memory - The CPU isolates the rightmost x bits of the register (according to the j value), acquires the operand word, inserts the desired partial word and replaces the operand word. The remainder of the operand word is *not changed*. (In some 1100 designs, this function may take place in the memory cabinet, but the function is the same.)

An example should make both kinds of transfer clear:

```
K1M1        +          1,-1        . GENERATES 000001777776
            L,H1       A5,K1M1     . GET LEFT HALF OF K1M1
            L,XH2      A6,K1M1     . GET SIGN-EXT RIGHT HALF
            S,H2       A5,K1M1     . REPLACE RIGHT HALF OF K1M1
```

Can you determine what will result in A5, A6 and K1M1, after these instructions are executed? Here are the results:

- A5 = 000000000001. The 1 in the upper (left) half of K1M1 was transferred to A5, zeroing out the rest of the register.

- A6 = 0777777777776. The -1 (minus 1) in the lower (right) half of K1M1 was transferred to A6. The sign bit of the lower half of K1M1 was extended into the upper half of A6.
- K1M1 = 000001000001. The rightmost 18 bits of A5 replace the right half of K1M1.

7.2.1 Quarter-Word and Third-Word j Values

The explanation of the remaining partial word designators has to bring in another hardware element: the designator register. This register contains a number of toggles and flags. For example, one of those flags controls whether an interrupt occurs upon a floating-point underflow condition. The designator register has had various names and formats in different 1100 implementations, but most of the function important to MASM programmers has stayed constant.

One of those flags determines the interpretation of j values 4, 5, 6 and 7. When this flag, called the Quarter-Word Mode Designator, is clear, the CPU interprets j values 4 through 7 as:

- XH1 (j=04) - The left half of the operand word, 18 bits, is transferred with sign extension.
- T1 (j=07) - The leftmost third of the operand word, 12 bits, is transferred with sign extension.
- T2 (j=06) - The middle third of the operand word, 12 bits, is transferred with sign extension.
- T3 (j=05) - The rightmost third of the operand word, 12 bits, is transferred with sign extension.

If the Quarter-Word Mode Designator is set, the CPU interprets j values 4 through 7 as:

- Q1 (j=07) - The leftmost quarter of the operand word, 9 bits, is transferred.
- Q2 (j=04) - The second quarter of the operand word is transferred.
- Q3 (j=06) - The third quarter of the operand word is transferred.
- Q4 (j=05) - The rightmost quarter of the operand word is transferred.

If this seems confusing, just remember that the default setting for the Quarter-Word Designator is clear (zero). That is, j values 4 through 7 mean XH1 and T1 through T3. You will have to take positive action to set quarter-word mode and have the CPU interpret those same j values as Q1 through Q4. The possible actions include a collector option, a collector directive, an executive request, or an LPD instruction (all discussed in Chapter 21).

One other point about the Quarter-Word Designator: Its bit position in the designator register has changed on each 1100 CPU. On the 1100/60, /70, and /80, it is bit D10. On the 1100/90, it is bit DB32. But the mechanisms used to set the quarter-word designator are system-independent.

7.2.2 Partial Word Examples

Assume this line has been coded previously:

```
LABEL       +           0123456765432    .
```

Below are the values that will result from loading various partial words from LABEL. For brevity, QWD=0 means the quarter-word mode designator is clear, and QWD=1 means it is set. (The hyphen below is a separator, not a sign.)

- W - 0123456765432 (that is, the entire word),
- H1 - 000000123456,
- H2 - 000000765432,
- T1 - (QWD=0) - 000000001234,
- T2 - (QWD=0) - 0777777775676 (leftmost of 12 bits was set, causing sign extension)
- T3 - (QWD=0) - 0777777775432 (sign extension again),
- Q1 - (QWD=1) - 000000000123,
- Q2 - (QWD=1) - 000000000456,
- Q3 - (QWD=1) - 000000000765,
- Q4 - (QWD=1) - 000000000432,
- S1 - 000000000012,
- S2 - 000000000034,
- S3 - 000000000056,
- S4 - 000000000076,
- S5 - 000000000054,
- S6 - 000000000032.

7.2.3 Partial Words Inoperative Between Registers

It may seem tempting to manipulate partial words between registers in this fashion:

```
L,H1        A5,X11      .
```

This does not work! (And MASM will not reject it, either.) The Processor and Storage Reference for each system states that when the u field is less than 0200, that is, a value coming from a register,

an entire word is always transferred. Don't make this mistake. Instead, you can transfer whole words and use shifts or other techniques to isolate the fields you need.

7.2.4 Immediate Operands

There remain two j values, 016 and 017. In the previous chapter, the examples used very simple operands, such as LABEL in this example:

```
LABEL    +        27
         L        A5,LABEL
```

A memory location like LABEL is just one kind of operand.

When the j field of an instruction is 016 or 017, the CPU uses a portion of the instruction word itself as the operand, rather than the contents of another memory location.

- U (j=016) - The CPU uses the 18-bit contents of the h, i and u fields as the operand.
- XU (j=017) - The CPU sign uses the sign-extended 18-bit contents of the h, i and u fields as the operand. XU is used most often for negative quantities.

Using immediate operands makes operations with constants easier, and less memory is required. This example from the previous chapter has been rewritten using U for a constant:

```
WORDS    +        2000
SECTORS  +        0
OFFSET   +        0
         L        A13,WORDS
         SZ       A12
         DI,U     A12,28
         S        A12,SECTOR
         S        A13,OFFSET
```

The SZ puts a 0 in A12. (Note that you can put a register in the u field.) The DI,U divides A12/A13 by 28.

You can also use immediate operands with AA, ANA, MSI, and many other instructions. Immediate operands do not work with any store instruction, including SZ, SFS, etc.

7.3 The $EQUF Directive

A $EQUF works much like a $EQU, except that it can incorporate several values describing a field:

- An offset (u). This is the part you would normally use a $EQU for.
- An index register (x).
- A partial-word modifier (j).

- The index register incrementation (h) and indirect (i) flags.

These should be familiar from the instruction word format. In fact, the instruction word fields defined by a $EQUF dovetail with the fields supplied on an instruction image to form a complete instruction word. This makes coding references to a field easy. Here is a sample $EQUF and instruction which uses it:

```
WRDCNT      $EQUF       4,X5,H1  .
            L           A1,WRDCNT  .
```

The $EQUF and load generate the same instruction word as:

```
            L,H1        A1,4,X5    .
```

By using the $EQUFed field, you save the work of coding the index register, offset, and partial word separately on each field reference. Using the $EQUF also reduces the chance of error. As you can see from the example, the subfields of the directive are offset, index register and partial-word modifier, in that order.

Each of the subfields on a $EQUF are optional. You can code a zero for omitted subfields, or code null subfields, and MASM will leave them zero. For example, these two lines omit the index register, and they are equivalent:

```
FIELD       $EQUF       4,,H1      .
.
FIELD       $EQUF       4,0,H1     .
```

In the first version, the commas serve as place holders for the index subfield.

Exercises

Use these data areas in the exercises which follow:

```
CELL01      +           17         .
CELL10      +           52,1028,1  .
WORD90      +           12,2       .
LOGLAS      +           42,92,4,136  .
LOG1ST      +           5,6,8,9,22,63  .
```

Write instructions to load the values below into A0. (Hint: what are the partial-word names for the parts each word is divided into?) Assume that quarter-word mode is set or cleared according to the partial word you have referenced.

E7-1. The 4 in LOGLAS.

The 1 in CELL10.

The 92 in LOGLAS.

The 9 in LOG1ST.

The 1028 in CELL10.

The 2 in WORD90.

E7-2. Write $EQUF directives to use the fields you referenced in E7-1, then recode the load instructions to use the $EQUFed fields. (Hint: none of the $EQUFs will use the index register subfield.)

E7-3. For each partial-word type, figure out the maximum number you can fit in that type. (Hint: don't forget sign extension on XHn and Tn reduces the number of significant bits.)

H1, H2

XH1, XH2

T1, T2, T3

Q1, Q2, Q3, Q4

S1, S2, S3, S4, S5, S6

Chapter 8: How To Alter Program Flow

8.1 Introduction to Jumps

The instructions described so far do not have any effect on the flow of control. If you needed to look at 10 consecutive entries in a table, you would need to code the same instruction 10 times. (Load the first entry into a register, add the second, add the third, and so forth.)

This chapter introduces instructions that let your program:

- Unconditionally transfer control to another location.
- Transfer control conditionally based on register or memory contents.
- Repeat sequences of instructions.

The same instructions can often be used to accomplish more than one of these goals. Their effect depends on the way they are used.

8.2 Unconditional Jumping

The J (Jump) instruction transfers control to the operand address. This instruction jumps to LOOP2:

```
                J           LOOP2  .
                .
                .
                .
LOOP2  .
```

(The dots between the jump instruction and the label LOOP2 stand for intervening instructions; they do not affect the operation of the jump.)

A jump instruction can jump forward, like the previous example, or backwards, as in this example:

```
LOOP1  .
                .
                .
                .
                J           LOOP1  .
```

The destination (operand) address in a jump *must* be inside your program. ("Inside your program" means within the range of addresses defined by the absolute element.) MASM will not reject a jump to zero, but trying to execute a jump to zero will result in an error.

STYLE: MASM will accept a jump to a label with an offset (for example, LABEL+5), and the jump will execute correctly. But such code is *bad practice*. If you insert instructions between LABEL and the intended destination of the jump, the jump will go to the wrong place. Or, you might have to modify all the places you jumped to LABEL+5. If you don't catch them all, the program will

---

not behave correctly. Do not scrimp on labels. MASM can handle thousands of labels; one or two more will not tax it.

STYLE: The way you place comment lines can help or hinder reading the code later on. For example, place a blank comment line between an unconditional jump and a label image:

```
            L       A0,STATUS .
            J       AWAY .
.
NOIO .
```

This practice has two benefits:

- It places a sharp demarcation at the end of a line of straight-through code.
- It isolates the label with "white space". Control can only pass to the label by a jump from elsewhere in the program.

If you follow this convention, you take advantage of your own process of perception, or that of someone who must support the code later.

Conversely, a label in the middle of a straight-through sequence of instructions should *not* be preceded by a blank comment line. Comments associated with the label (such as what conditions cause a jump there) should be on the same line as the label.

8.3 Conditional Jump Instructions

The next group of jumps operate depending on the contents of an A register. The circumstance in which they jump is given in the name. Here are the first two:

- JZ (Jump Zero) - jumps to the destination address if the named A register (A_a) is plus or minus zero. Otherwise, control will fall to the next instruction.
- JNZ (Jump Nonzero) - jumps to the destination address if the named A register (A_a) is other than plus or minus zero. Otherwise, control will fall to the next instruction.
- DJZ (Double Jump Zero) - jumps to the destination address if both A_a and A_{a+1} are both zero. Otherwise, control will fall to the next instruction.

(These descriptions refer to the "named A register". By this is meant the "contents of the named A register". From here on, a reference to a register will mean the *contents of the register* unless otherwise noted. Likewise, the term "named A register" will be replaced by "A_a" from here on.)

Here is an example use of JZ:

```
            AN          A0,A9       .
            JZ          A0,NXTFLD   .
            .
            .
            .
NXTFLD      .
```

This example subtracts register A9 from register A0. If the result is zero, the JZ instruction will jump to NXTFLD. If the result is nonzero, control will pass to the instruction following the JZ.

Conditional jumps follow this rule:

- If the condition for the jump is not satisfied (nonzero on a JZ, for example), control passes to the following instruction.

This rule will not be restated in the descriptions of the remaining conditional jumps.

Here is an example use of JNZ:

```
CYCLE       +           0           .
            L           A15,CYCLE   .
            JNZ         A15,INVALD  .
            .
            .
            .
INVALD      .
```

This example loads A15 from CYCLE. If that value is nonzero, control passes to the label INVALD.

This is a good time for you to pull out your instruction summary table (in your Quick-Reference Guide or Processor and Storage Manual). Locate J, JZ, and JNZ in the table and familiarize yourself with the descriptions found there. (You may have to use the alphabetic Mnemonic Instruction/Function Code Cross Reference table; the instruction summaries are in function code order.) As you read the descriptions of other instructions, refer to the instruction summary table.

"U" in the instruction summaries for J, JZ, and JNZ refers to the destination address. The next chapter will explain U in depth.

The next pair of instructions is:

- JP (Jump Positive) - jumps to the destination address if A_a is positive (that is, the sign bit is clear).
- JN (Jump Negative) - jumps to the destination address if A_a is negative (that is, the sign bit is set).

Your instruction summary notes that these instructions test bit 0 (35), the sign bit. JP and JN ignore everything else in A_a.

This example shows the use of JN and a partial word with sign extension:

```
FLAG1       -           5,0,157     .
            L,T1        A10,FLAG1   .
            JN          A10,LABEL2  .
            .
            .
            .
LABEL2  .
```

Given the contents of FLAG1 as shown, the result of the L,T1 instruction will be minus 5 (0777777777772). Since that will set the sign bit in A10 (the leftmost bit), the JN will jump to LABEL2.

Suppose the partial word on the load instruction was H1 instead of T1. Will the JN jump in this case? (Hint: figure out what the load will put into A10. Will it set the sign bit? H1 will not do sign extension.)

Note that JP will jump on a value of plus 0 (all 0 bits), and JN will jump on a value of minus 0 (all 1 bits).

The final pair of conditional jump instructions test the opposite end of a register from JP and JN. JB (Jump Low Bit) and JNB (Jump No Low Bit) test bit 35 (0), the rightmost bit. Here are their definitions:

- JB jumps to the destination address if the rightmost bit of A_a is set (1).
- JNB jumps to the destination address if the rightmost bit of A_a is clear (0).

This example shows the use of JB:

```
FLAG2       +           3,0 .
            L,H1        A7,FLAG2  .
            JB          A7,ZZ2010  .
            .
            .
            .
ZZ2010  .
```

This example shows the use of JNB:

```
FLAG1       +           5,0 .
            L,H1        A11,FLAG1  .
            JNB         A11,LABEL2  .
            .
            .
            .
LABEL2  .
```

The JNB instruction will *not* jump in this example. The load instruction will put a 5 in A11. Recall

the binary value of 5, 1012. Since the rightmost bit of 5 is a 1, the JNB will "fail" and fall through to the next instruction.

Since the binary equivalent of 3 is 112, the JB will jump. The rightmost bit of 3 is a 1 bit. (Hint: the rightmost bit of odd numbers is always set; they are not evenly divisible by 2.)

Note that all 6 conditional jumps discussed so far test the contents of an A register.

8.4 Test Instructions Which Don't Need Registers

Instead of looking at just the contents of a CPU register, the test instructions test the contents of an operand address. That operand can be a register (X, A, or R) or a memory location.

- If the test is satisfied, the test instruction *skips* the following instruction.
- If the test is not satisfied ("fails"), the following instruction is executed.

(Your instruction summary tables and this text refer to the instruction following a test as NI - Next Instruction.)

The instruction following a test instruction may be any instruction. In many cases, you will choose to put a jump following a test. The jump will take place if the test fails. At other times, you may need to code a load, store, store zero or even a conditional jump.

The first two pairs of test instructions do not require registers. They are:

- TZ (Test Zero) - if the operand is plus or minus zero, the following instruction is skipped.
- TNZ (Test Nonzero) - if the operand is anything other than plus or minus zero, the following instruction is skipped.
- TP (Test Positive) - if the sign bit of the operand is clear (0), the following instruction is skipped.
- TN (Test Negative) - if the sign bit of the operand is set (1), the following instruction is skipped.

This example shows the use of TZ:

```
COUNT5      +           3,0          .
            SZ          A0           .
            TZ          COUNT5       .
            J           BUMP1.
            L           A0,COUNT5    .
            .
            .
            .
BUMP1   .
```

What will be the result in A0? Where will control pass to? Control passes to BUMP1, and A0 will

be zero. That is because the TZ instruction will "fail"—its operand nonzero—and execute the next instruction, a jump to BUMP1. Since the SZ zeroed A0, that zero will still be in A0 when control passes to BUMP1.

Here is a TNZ example:

```
TALLY       +           115,0       .
            TNZ,H1      TALLY       .
            J           ZEROIT      .
            L           A12,TALLY   .
            .
            .
            .
ZEROIT      .
            SZ          A12         .
```

In this example, control will continue after the TNZ instruction with the L A12,TALLY.

This example uses TP, Test Positive:

```
COUNTER     +           115         .
            TP          COUNTER     .
            J           AAA099      .
            L           A1,COUNTER  .
            .
            .
            .
AAA099      .
            SZ          A1          .
```

COUNTER is positive (the sign bit is clear), so the J AAA099 is skipped. Execution proceeds with the L A1,COUNTER.

Here is an erroneous code fragment using TN, Test Negative. Can you spot the error?

```
FLAG1       -           5,0         .
            TN,H1       FLAG1       .
            J           LABEL2      .
            LN          A0,FLAG1    .
            .
            .
            .
LABEL2      .
```

The TN will "see" an operand of 000000777772, therefore the sign bit will be clear. This effect was probably not intended. If you want to test a partial-word quantity which can go negative, be sure to ask for sign extension. Change the H1 to XH1:

```
FLAG1       -           5,0          .
            TN,XH1      FLAG1        .
            J           LABEL2       .
            LN          A0,FLAG1     .
            .
            .
            .
LABEL2      .
```

In this corrected version, the TN will test the sign-extended H1 of FLAG1, seeing 0777777777772. The sign bit is set, thus the test is satisfied, and the TN skips the next instruction.

Although the test instruction examples so far have tested a memory cell and jumped on test failure, there are other options:

- A test instruction can test a register.
- A test instruction can be followed by an instruction other than a jump.

This example shows both possibilities:

```
BUFF1       RES         112          .
BUFF2       RES         112          .
            L           R6,BUFF1+7   .
            TNZ         R6           .
            L           R6,BUFF2+7   .
```

The first instruction puts the contents of the eighth word of BUFF1 in R6. If that word is zero, the second load instruction will get the contents of the eighth word of BUFF2. (Remember, BUFF1+0 is the first word, BUFF1+1 is the second and so forth.)

8.5 Test Instructions Which Do Need Registers

Test instructions in another group compare an operand with an A register. The instruction name (for example, Test Equal) includes a relation ("Equal"). If that relation between operand and register is true, the test skips the next instruction. The operand may be a memory location, or it may be another register. Here are the instructions:

- TE (Test Equal) - if the operand equals A_a, the next instruction is skipped.
- TNE (Test Not Equal) - if the operand does not equal A_a, the next instruction is skipped.
- TLE (Test Less Than Or Equal) - if the operand is less than or equal to A_a, the next instruction is skipped.
- TG (Test Greater Than) - if the operand is greater than A_a, the next instruction is skipped.
- DTE (Double Test Equal) - if the contents of U and U+1 equal the contents of A_a and A_{a+1} respectively, the next instruction is skipped.

This is the general form of these descriptions:

- If operand <relation> A_a, skip NI.

Remembering this general form will help keep these tests straight in your mind.

Here is an example using TE:

```
WORK01      +           0,22        .
WORK02      +           15          .
            L,H2        A7,WORK01   .
            AN,U        A7,7        .
            TE          A7,WORK02   .
            J           XRM050      .
            MSI,U       A7,7        .
            .
            .
            .
XRM050      .
```

Since 22 minus 7 equals 15 (the result in A7), and WORK02 contains 15, the TE will skip the jump instruction. The next instruction executed will be the MSI.

You can also use TE and TNE to compare a register to a constant. The following example uses TNE. It will jump to MSS100 if the contents of MASK equals 0101 (octal 0101 equals 65 decimal).

```
MASK        +           63          .
            L           A4,MASK     .
            A,U         A4,1        .
            TNE,U       A4,0101     .
            J           MSS100      .
            A,U         A4,2        .
            .
            .
            .
MSS100      .
```

Since 63+1 equals 64 (0100 octal), the test will skip the jump instruction.

Note that in order to jump on equality (J MSS100 in the example), you need to use a Test *Not* Equal instruction. If A_a equals the operand, the following instruction is not skipped, but executed.

There are 2 ways to jump when an A register is nonzero. This is the hard way:

```
            TE,U        A13,0       .
            J           NOTZERO     .
```

This is the easy way:

```
            JNZ         A13,NOTZERO     .
```

Similarly, there are 2 ways to jump when an A register is zero. This is the hard way:

```
        TNE,U       A14,0       .
        J           ZERO        .
```

This is the easy way:

```
        JZ          A14,ZERO    .
```

Can you see the price paid by the hard way?

- First, the CPU must transfer the immediate operand, zero, into the arithmetic section. A conditional jump (JNZ or JZ) directly tests the register for zero.
- Second, the hard way uses an extra instruction word. (One of the goals in writing MASM is to minimize use of memory and CPU time. Besides, who wants to code 2 instructions when 1 will do?)

On the other hand, a test instruction with a register operand is handy to replace a missing value with a default:

```
PARAM1  +           0           .
        L           A8,PARAM1   .
        TNZ         A8          .
        L,U         A8,25       .
```

Since loading A8 from PARAM1 will get a zero, the test will fail. The zero in A8 will be replaced by a default, 25.

The last two tests listed above, TLE and TG, can be the hardest to understand and use. Once again, remember the "operand <relation> A_a" rule to get back on track.

Continuing from the last example, suppose that there was an upper limit of 50 on PARAM1. This example uses TG to replace the value in A8 with the maximum:

```
PARAM1  +           0               .
MAXPAR1 EQU         50              .
        L           A8,PARAM1   . GET PARAM1
        TG,U        A8,MAXPAR1+1  . EXCEEDS LIMIT?
        L,U         A8,MAXPAR1  .   YES,SET_TO_LIMIT
```

Here is how the TG works in this example:

- If PARAM1 is 50 or less, the test is successful and skips the next instruction. (By the "operand <relation> A_a" rule, 51 > 50.)
- If PARAM1 is 51 or more, the test fails; NI is executed, replacing the value in A8.

Note that you may have to adjust the test value with a constant, as in this example (MAXPAR1+1) to make a test work. If the operation of TG is not clear, try writing down possible values and plug them into this example to see how they behave.

STYLE: $EQUating the maximum value of PARAM1 to the label MAXPAR1 makes it easier to change when necessary. Each time you would code the limit value, code MAXPAR1 instead. Place parameters like this in a group near the beginning of your program to make them easy to find.

Continuing with the tests of PARAM1, assume that there is also a lower limit of 10. This sequence will replace a value below 10 with 10:

```
PARAM1      +          5
MINPAR1     EQU        10
            L          A8,PARAM1   . GET PARAM1
            TLE,U      A8,MINPAR1  . .LT. LOWER LIMIT?
            L,U        A8,MINPAR1  .  YES,SET_TO_LOWER_LIMIT
```

Once again, try plugging various values into PARAM1 to see how the TLE instruction behaves in this case. And again, when you code your own TLE instructions, you may need to add or subtract a constant to the test value to give the desired results.

The introduction to this chapter promised unconditional jumps, conditional jumps and the ability to repeat sequences of instructions. The techniques used to repeat a sequence are:

- Initialize a counter.
- Increment the counter on each repetition.
- Test the counter. If the maximum value has not been reached, jump to repeat the loop. If the maximum has been reached, exit the loop.

Those are the basics. But there is often an intimate connection between the loop counter and table addressing. Table addressing is explained in Chapter 9. Repeated sequences or *looping* deserves its own chapter, Chapter 10.

8.6 Within-Range Tests - TW, TNW

The last two test instructions test the contents of U against two adjacent A registers. Here are their descriptions:

- TW (Test Within Range) - if the contents of U are greater than A_a and less than or equal to A_{a+1}, the next instruction is skipped.
- TNW (Test Not Within Range) - if the contents of U are less than or equal to the contents of A_a or greater than the contents of A_{a+1}.

Note that the tests done by each instruction are asymmetrical, that is the TW does not test for $(A_a)<(U)<(A_{a+1})$; there is an "or equal" thrown in, and it is a possible source of confusion. This example shows how to test for the days of the month 15 through 22 inclusive:

```
LOWER     EQU       15-1              . LOWER LIMIT
UPPER     EQU       22                . UPPER LIMIT
DAYTAB    RES       10                . DAY TABLE
DAYMO     EQUF      DAYTAB+5,,S3      . FIELD TO TEST
.
          L,U       A9,LOWER          . LOWER LIMIT FOR TEST
          L,U       A10,UPPER         . UPPER LIMIT FOR TEST
.
          TW        A9,DAYMO          . WITHIN DATE LIMITS?
          J         P12090            . NO,P12090  YES,GO_ON
```

In other words, the limits you test with must be adjusted slightly for the types of tests done by the instruction (15 in the definition of LOWER).

By design, the TNW instruction performs exactly the opposite tests as TW. Therefore, if you want to jump when the test value passes the limits test, rather than skipping NI, you can use the very same A_a and A_{a+1} settings, as in this revision of the example above:

```
LOWER     EQU       15-1              . LOWER LIMIT
UPPER     EQU       22                . UPPER LIMIT
DAYTAB    RES       10                . DAY TABLE
DAYMO     EQUF      DAYTAB+5,S3       . FIELD TO TEST
.
          L,U       A9,LOWER          . LOWER LIMIT FOR TEST
          L,U       A10,UPPER         . UPPER LIMIT FOR TEST
.
          TNW       A9,DAYMO          . WITHIN DATE LIMITS?
          J         P12090            . YES,P12090  NO,GO_ON
```

STYLE: You could accomplish the same job as a TW or TNW by using two ordinary test instructions, but TW and TNW have these advantages:

- One TW/TNW with the associated jump takes only two memory locations, where two test instructions with associated jumps would take four. In other words, TW or TNW use half the memory.
- TW/TNW accomplish the same tests with only one data transfer from memory to CPU.

STYLE: To save time in execution, try to set up the test values for a TW outside a loop.

Exercises

E8-1. Assume H1 of the cell TRAB89 contains a flag. Code two different sequenceswhich will jump to PH1090 if that field is non-zero.

E8-2. Assume XH2 of the cell LIMITS has a signed number. Code two different sequences which will jump to P4U020 if that field is negative.

E8-3. Code a sequence to jump to MAIN50 if the lowest bit of A8 is set. How would you change it to jump if the lowest bit is clear?

E8-4. For "extra credit" (using information from a later chapter), can you use another test instruction to code an alternative solution to E8-3?

E8-5. Code a sequence which will jump to POST99 if the whole word TRACKS is greater than H2 of LIMIT.

E8-6. Repeat E8-5, assuming that LIMIT is a constant. (Hint: use an immediate operand.)

E8-7. Code a sequence to test the whole word PARAM7. If it is greater than or equal to the $EQUed value MAXBOX, set PARAM7 to 0.

Chapter 9: Extending Flexibility: Indexing, Indirect Addressing and Literals

9.1 Basic Indexing: XM

Your instruction summary charts refer to an operand address as "U" (the capital letter). But the instruction format diagram refers to an address field as a lower case "u". How does u get to be U? What goes on in the hardware?

Understanding the answer requires a moment's study of X registers. Each X register has two portions:

- An increment portion, the left half.
- A modifier portion, the right half.

When an instruction's x field is nonzero, the CPU adds the modifier part of the named X register (X_x) to u, creating U. If the instruction's x field is zero, u becomes U directly.

How do you request indexing on an instruction? Place the index register mnemonic in the index subfield of the operand field, like this:

```
L       A5,TABLE,X10
```

Read this instruction as "load A5 from TABLE indexed by X10". If TABLE's address at execution time is 040010, and X10's modifier is 024, then U will be 040034 (040010+024).

(The subfield used for indexing varies among instructions. On instructions which use an A register, the index subfield is third. On instructions which do not use an A register, the index subfield is second.)

Here are a few symbols you need to know:

- X_x, already discussed, means the X register identified by the x field of an instruction.
- XI means the increment portion of an index register; $XI(X_x)$ the increment portion of X_x.
- XM means the modifier portion of an index register; $XM(X_x)$ the modifier portion of X_x.

For example, $U = u + XM(X_x)$. Your instruction summary chart and this text will use this notation to show concisely how some operations work.

For the moment, focus on use of the modifier portion. Using the increment portion requires understanding the basics of indexing first.

What are some ways to use indexing? Here are a few:

- Addressing fields of a table of data.
- Jumping to one of a number of addresses depending on a variable.
- Testing the contents of an index register.
- Loading and storing characters. (This uses tables of load and store instructions. See Chapter 21.)
- Returning from a subroutine. (See Chapter 14.)

The last two uses of indexing are advanced topics, to be discussed later.

9.2 How To Address Fields in a Data Table

Before this chapter, all the examples used single data variables (*scalars*, in Fortran terminology). Many of the jobs done by MASM programs must work on repeated data, such as processor call specifications, multiple disk units, multiple files, and so forth. It is not practical to code an identical sequence of instructions for each entry in a table. Therefore, the same sequence of instructions is used in a loop, and an index register addresses each entry in turn.

Here is a simple loop. It counts the nonzero entries in the table SCORES:

```
SCORES  .
            +           5  .
            +           0  .
            +           1  .
            +           0  .
            SZ          A0          .  INITIALIZE COUNTER
            SZ          A3          .  INITIALIZE INDEX
LOOP1       .
            TZ          SCORES,A3   .  ENTRY NONZERO?
            A,U         A0,1        .   YES,BUMP   NO,
            A,U         A3,1        .  INCREMENT INDEX .
            TLE,U       A3,4        .  INDEX REACHED MAX?
            J           LOOP1       .   NO,LOOP1   YES,
            .
```

Notice this example uses the highest-numbered index register, A3. (Its x field value is 017; all four bits are set.) Follow the comments to understand what each instruction does. Remember that if the operand of a TZ (Test Zero) instruction is nonzero, NI is executed. If zero, NI is skipped. If SCORES indexed by A3 is nonzero, the A,U A0,1 is executed, incrementing the nonzero entry count. The first time through the loop, A3 is zero; the TZ tests the first entry of SCORES. The second time, A3 is 1, the TZ tests the second entry, and so forth.

How will this sequence stop? The TLE,U will compare against, successively, 1, 2, 3 and 4. In the first 3 cases, the TLE will "fail", executing the next instruction, the jump. (That is, 4 is not less than or equal to 1, 2 or 3). In the fourth case, the TLE will succeed (4 is less than or equal to 4) and skip the jump, exiting the loop.

Suppose the number of entries in SCORES changed. What effect would that have on instructions such as the TLE,U? A previous style box referred to the situation, and here is a solution:

```
SCORES  .
        +       5  .
        +       0  .
        +       1  .
        +       0  .
        +       7  .
NSCORE  EQU     $-SCORES  .
        SZ      A0        . INITIALIZE COUNTER
        SZ      A3        . INITIALIZE INDEX
LOOP1   .
        TZ      SCORES,A3 . ENTRY NONZERO?
        A,U     A0,1      .  YES,BUMP  NO,
        A,U     A3,1      . INCREMENT INDEX .
        TLE,U   A3,NSCORE . INDEX REACHED MAX?
        J       LOOP1     .  NO,LOOP1  YES,
        .
```

STYLE: Note the constant NSCORE. The construct $-SCORES generates the number of words (entries) in the table. In this case, it is 5. It will change as the length of the table changes (as it did in this version). *It is impossible to overstate how much work this one technique will save!* Note also that the test instruction now tests A3 against the value NSCORE instead of the constant 4.

Of course, table entries are seldom exactly one word. We may try to squeeze 2 or more fields into each word, or divide multiple words into different-sized fields. As an example of the latter, assume that each SCORES entry contains three fields. Two fields may have values up to 250,000, and the third, over 500,000. (Hint: these values affect the required bit width of the fields.)

Each table entry needs 2 words. Two fields go into halves of the first word; the other, with values over 500,000, goes into a full word.

Here is the *worst* way to solve the problem of tallying the nonzero values of each of the 3 fields:

```
SCORES  .
        +       120567,100435 . TWO FIELDS
        +       477522       . SINGLE FIELD
        +       50,200123    . TWO FIELDS
        +       35204        . SINGLE FIELD
        +       12489,78905  . TWO FIELDS
        +       1356078      . SINGLE FIELD
        SZ      A0           . INITIALIZE COUNTER
        SZ      A1           . INITIALIZE COUNTER
        SZ      A2           . INITIALIZE COUNTER
        SZ      A3           . INITIALIZE INDEX
```

```
LOOP1 .
            TZ,H1       SCORES,A3   . FIRST FIELD NONZERO?
            A,U         A0,1        .  YES,BUMP
            TZ,H2       SCORES,A3   . SECOND FIELD NONZERO?
            A,U         A1,1        .  YES,BUMP
            TZ          SCORES+1,A3 . THIRD FIELD NONZERO?
            A,U         A2,1        .  YES,BUMP
            A,U         A3,2        . INCREMENT INDEX .
            TLE,U       A3,6        . INDEX REACHED MAX?
            J           LOOP1       .  NO,LOOP1  YES,
            .
```

(Actually, the worst way would omit the comments, too!)

What is wrong with this code? Suppose the entries had to be made bigger. Assume you had to expand one of the half-word fields or add another field, for example. The problem of chasing down constants would be made worse still; you'd have to change a bunch of partial-word modifiers, too. (The author had exactly this problem on his first major project.) The solution to this problem uses the MASM directive $EQUF (EQUate Field, pronounced EE-quahf), described in Chapter 7.

The following revision of the example is a major improvement. In fact, it makes the difference between a program which is maintainable and one which is not:

```
FIELDA      $EQUF       SCORES+0,,H1 .
FIELDB      $EQUF       SCORES+0,,H2 .
FIELDC      $EQUF       SCORES+1,,W .
SCRLEN      $EQU        2           . LENGTH OF AN ENTRY
SCORES      .
            +           120567,100435 . TWO FIELDS
            +           477522      . SINGLE FIELD
            +           50,200123   . TWO FIELDS
            +           35204       . SINGLE FIELD
            +           12489,78905 . TWO FIELDS
            +           1356078     . SINGLE FIELD
NSCORE      $EQU        ($-SCORES)/SCRLEN . NBR OF ENTRIES
            SZ          A0          . INITIALIZE COUNTER
            SZ          A1          . INITIALIZE COUNTER
            SZ          A2          . INITIALIZE COUNTER
            SZ          A3          . INITIALIZE INDEX
LOOP1 .
            TZ          FIELDA,A3   . FIELDA NONZERO?
            A,U         A0,1        .  YES,BUMP  NO,
            TZ          FIELDB,A3   . FIELDB NONZERO?
            A,U         A1,1        .  YES,BUMP  NO,
            TZ          FIELDC,A3   . FIELDC NONZERO?
            A,U         A2,1        .  YES,BUMP  NO,
            A,U         A3,SCRLEN   . INCREMENT INDEX .
            TLE,U       A3,SCRLEN*NSCORE . INDEX AT MAX?
            J           LOOP1       .  NO,LOOP1  YES,
            .
```

This example omits the index register from the $EQUF directives (perfectly acceptable and sometimes necessary).

By the way, the machine does not care how the program arrives at a table address. This example so far puts the offset from SCORES into A3.

- In cases where you want to maintain parallel tables—the same index value can be used for both—that works fine.
- In other cases, you may want the index register to point to (contain the complete address of) a given table entry.

Either way, the U address which results at execution time is the same. This variation on the previous example uses the latter technique:

```
FIELDA    $EQUF     0,,H1       .
FIELDB    $EQUF     0,,H2       .
FIELDC    $EQUF     1,,W        .
SCRLEN    $EQU      2           . LENGTH OF AN ENTRY
SCORES    .
          +         120567,100435 . TWO FIELDS
          +         477522      . SINGLE FIELD
          +         50,200123   . TWO FIELDS
          +         35204       . SINGLE FIELD
          +         12489,78905 . TWO FIELDS
          +         1356078     . SINGLE FIELD
NSCORE    EQU       ($-SCORES)/SCRLEN . NBR OF ENTRIES
          SZ        A0          . INITIALIZE COUNTER
          SZ        A1          . INITIALIZE COUNTER
          SZ        A2          . INITIALIZE COUNTER
          L,U       A3,SCORES   . INITIALIZE INDEX
LOOP1 .
          TZ        FIELDA,A3   . FIELDA NONZERO?
          A,U       A0,1  .  YES,BUMP  NO,
          TZ        FIELDB,A3   . FIELDB NONZERO?
          A,U       A1,1  .  YES,BUMP  NO,
          TZ        FIELDC,A3   . FIELDC NONZERO?
          A,U       A2,1  .  YES,BUMP  NO,
          A,U       A3,SCRLEN   . INCREMENT INDEX .
          TLE,U     A3,SCRLEN*NSCORE+SCORES . MAX?
          J         LOOP1 .  NO,LOOP1  YES,
          .
```

A3 now contains the *address* of each entry, not its offset from SCORES. Note that the $EQUFs now contain offsets from the beginning of the *entry*, not an address. (If necessary, compare this and the previous example side by side. Be sure you understand how this variation differs from the previous one.)

The way you code $EQUFs versus index register values can depend on several factors:

- If you are coding a linear table, like the ones shown, you have a choice. You can put an

address in each $EQUF like the first example, $EQUFing FIELDA to SCORES+0,,H1, or put the address in the index register, $EQUFing FIELDA to 0,,H1. *For a given table, you must do it one way or the other; you must be consistent.* The way you define the index register value determines the way you test it; that is, including the address of the start of the table in the test value or not. (Refer to the TLE,U in the examples to see how the test changed.)

- The simplest way to code pointers that join linked list nodes puts the actual address of the next node into the pointer field. In this case, the $EQUFs should refer to the beginning of the node (that is, table entry), rather than the beginning of the buffer. The $EQUFs should be like the most recent example (0,,H1; 0,,H2; and so forth). (If you are not familiar with data structures, do not fret; they will be covered in chapter 20.)

- Tables used by certain Executive Requests often contain pointers relative to the start of the buffer. If you load such a pointer into an index register, you must either add the start address of the buffer yourself, or include the buffer start address in each $EQUF. (For the best example of this type of Exec-returned data structure, turn to the MSALL$ function of ER MSCON$ in the Installation Reference.)

9.3 Index Register Incrementation: XI and h

The beginning of this chapter referred to the two halves of each index register: the left, or increment portion, and the right, or modifier portion. The modifier portion of the X_x index register is added to u to make U, the effective operand address.

We turn now to the top half of an index register, XI. If we set the h field of an instruction to 1, the CPU will take the increment portion of X_x and add it to the modifier portion of X_x, placing the result back in the modifier portion. In more concise notation, $XM(X_x) = XM(X_x) + XI(X_x)$. XI may be negative, effectively subtracting from XM.

While at first, this may seem like an arcane maneuver, in fact, it can save an instruction on each pass through a loop. If we set XI to the size of a table entry, bumping to the next entry of a table is no more effort or overhead than putting an asterisk on the appropriate operand subfield:

```
        TZ          FIELDC,*A3        . FIELDC NONZERO?
```

The asterisk (*) on A3 sets the h field of the instruction word. To make this technique work requires planning and coordination of these elements:

- The index register's increment field must be set up with the table entry size (in the current example, 2).

- You must be able to identify the loop's last reference to any field of the current entry (in the current example, the test of FIELDC), and set the h field of that instruction.

- Do not explicitly add the entry size to the index register any more; the hardware will do it for you.

Note the instructions changed from the previous example:

```
FIELDA      $EQUF       0,,H1       .
FIELDB      $EQUF       0,,H2       .
FIELDC      $EQUF       1,,W        .
SCRLEN      $EQU        2           .
SCORES      .
            +           120567,100435 . TWO FIELDS
            +           477522      . SINGLE FIELD
            +           50,200123   . TWO FIELDS
            +           35204       . SINGLE FIELD
            +           12489,78905 . TWO FIELDS
            +           1356078     . SINGLE FIELD
NSCORE      EQU         ($-SCORES)/SCRLEN . NBR OF ENTRIES
            SZ          A0          . INITIALIZE COUNTER
            SZ          A1          . INITIALIZE COUNTER
            SZ          A2          . INITIALIZE COUNTER
            L,U         A3,SCORES   . INITIALIZE INDEX
            LXI,U       A3,SCRLEN   . SET INCREMENT (TABLE SIZE)
LOOP1       .
            TZ          FIELDA,A3   . FIELDA NONZERO?
            A,U         A0,1        .  YES,BUMP  NO,
            TZ          FIELDB,A3   . FIELDB NONZERO?
            A,U         A1,1        .  YES,BUMP  NO,
            TZ          FIELDC,*A3  . FIELDC NONZERO?
            A,U         A2,1        .  YES,BUMP  NO,
            L,U         A4,0,A3     . ISOLATE MODIFIER PORTION
            TLE,U       A4,SCRLEN*NSCORE+SCORES . MAX?
            J           LOOP1       .  NO,LOOP1  YES,
            .
```

As in this example, index register incrementation is most useful stepping through the entries of a sequential table. You will doubtless find other uses for it.

Tip: if you cannot identify the single last reference to a table entry, use an instruction not discussed yet, NOP, or No OPeration. This instruction does nothing, normally. But you can coax it into performing your index register incrementation like this:

```
            NOP         0,0,*A3  .
```

In other words, supply both the index register number and the h bit, and the CPU will cheerfully increment your index register for you. The instruction will have no other effect. Later in this chapter, you will see other uses of NOP.

(The L,U in the shaded area is an example of a technique described in the next section.)

9.4 Using an Index Register With Immediate Operands

One important use of indexing is modifying immediate operands. Recall that U is formed by adding u and XM(X_x). And putting a partial word-designator of U on an instruction places U into the named register. You can use a combination of u and XM(X_x) to transfer portions of an index register to another register, such as this example does:

```
L,U         A1,0,A0     .
```

The modifier portion of A0 will be added to 0 (u), and the sum placed in A1. Note that the contents of the upper half of A0, the increment portion, are ignored. Similarly, you can test the lower half of an index register by itself:

```
TZ,U        0,X5        .
J           RTZ010      .
```

If the lower half of X5 is zero, the jump is skipped; if nonzero, the jump is taken. You can also use this technique to create an address for an ER. Suppose the ER packet is in the middle of a table entry pointed to by X10. An ER will require the actual address of the packet; how do you get it? Use code like this:

```
L,U         A0,PACKET,X10 . GET PACKET ADDRESS IN A0
ER          <name>        . DO THE ER
```

If the ER also needs a value in the left half of A0, use an LXI to fill it in:

```
LXI,U       A0,PKTLEN   .
```

9.5 LXM and LXI - Special Instructions for Index Registers

Load X Modifier (LXM) and Load X Increment (LXI) load only their respective portions of index registers, without affecting the other halves. LXM loads the contents of U into the modifier portion of X_a, without affecting the increment portion. LXI loads the contents of U into the increment portion of X_a, without affecting the modifier portion.

STYLE: Before using these instructions, however, remember what a simple LX does. An "LX,U X5,TABLE" puts the address of TABLE into the modifier portion of X5 and zeroes the increment portion of X5. Don't use an LXM to put the address of TABLE into X5 unless you really mean to preserve the increment portion of the register. That kind of usage can confuse someone attempting to follow your thinking through your code. The same advice goes for LXI, too.

9.6 Indirect Addressing

Chapter 3, in covering the instruction word format, mentioned the indirect (i) bit. If this bit is set in an instruction, the CPU treats the effective operand address (U) not as the location of an operand, but part of another instruction word. It obtains the rightmost 22 bits of that word (pointed to by U), and calculates another effective operand address. (Recall that those 22 bits contain the x, h, i and u fields. The CPU adds u and XM(X_x) to get this new operand address.) (Refer to Figure 9-1 on the following page for an example.)

Normally, the process stops right there. The CPU goes to the new U to obtain the operand for the original instruction. If, however, the i bit obtained along with the x and u fields is set, the process continues over and over again until the CPU obtains x, h, i and u without the i set. Repetition of that process beyond the first time is called *cascading* indirect references, and is fairly rare. (The hardware

will allow you to cascade indirect references more deeply than you may need.) In fact, actual use of indirect addressing is rare enough; but covered here in case you may encounter it someday.

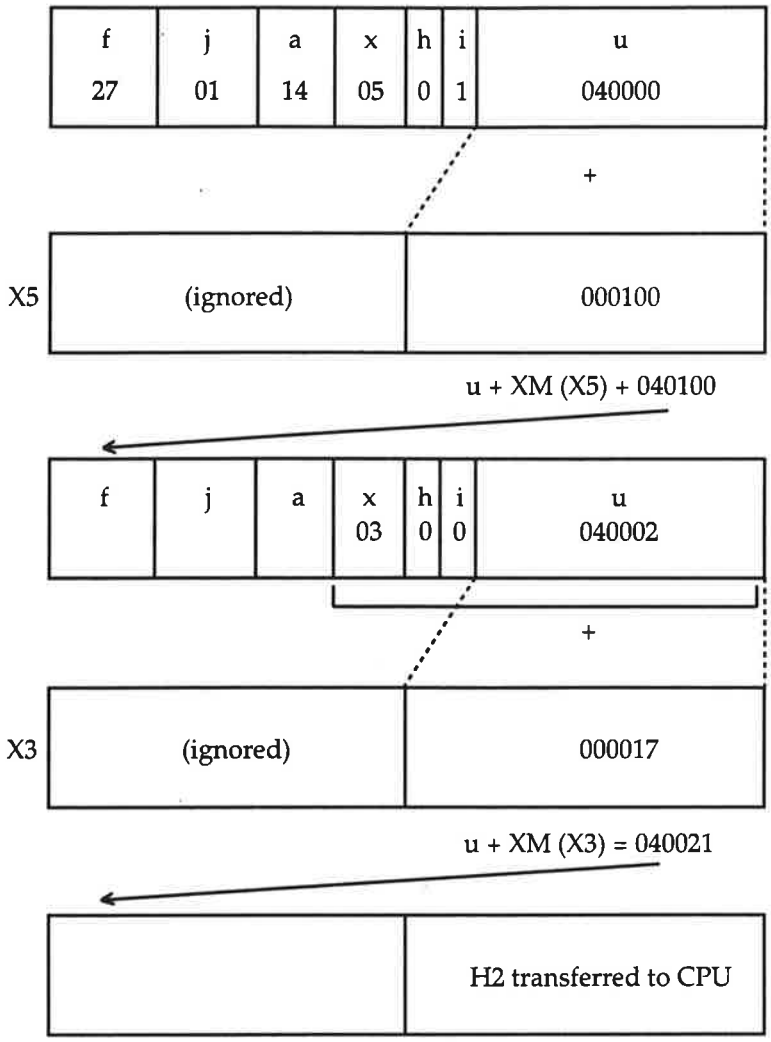

Figure 9-1. Indirect Addressing

Here are two uses of indirect addressing:

- Returning from an SLJ instruction. SLJ (Store Location and Jump) stores the jump-from address in U and jumps to U+1. When the routine called by the SLJ needs to return, it can jump indirect on the first word of the routine. (More on SLJ in Chapter 14.)
- A jump table. (Also done better with index registers, due to their full 18-bit width.)

Note that extended mode (available on the 1100/90 and the 2200s) does not support indirect addressing.

9.7 Literals: Easy Constants

MASM is equipped to do a useful bit of housekeeping for you. Sometimes, you need to code a constant, or even a whole instruction, for a single use. Normally you might want to code a separate line with a plus '+' operator, or an $EQU directive, and an explanation of the constant's use, like these:

```
K1M1        +           1,-1        . BEGINNING INDEX
ENTSIZ      $EQU        12          . TABLE ENTRY SIZE
```

But suppose you wanted to set up an index register one time with the contents of K1M1, without the work of a whole separate code line. (This assumes the constant is of minor importance.) MASM will let you code the operation and operand portions of a code line in parentheses in the u field, like this:

```
            L           X5,(+ 1,-1)     .
```

MASM allocates memory for the constant (called a *line item*) and uses that address in the u field of the instruction.

You can put almost anything into a literal:

- Multiple values for partial words, like (+ 14,5,2),
- A $FORM name (described in the next section) like (PF 12,1,BUFFER), or
- A character string, like ('PLEASE MOUNT TAPE F27').

Here are examples using a double-word constant and a jump instruction:

```
            DL          A1,('MYFILENAME  ') .
            L           A0,(J CPG020)  .
```

The $LIT directive controls the placement of literals. Here is its format:

```
$(3)        $LIT                    .
```

The location counter may be any you choose, based on where you want MASM to put the generated data. The named location counter also becomes active until another is used. (Hint: your choice of an odd or even counter determines whether the data will end up in an instruction bank or a data bank.)

- If there is no $LIT directive, MASM places literals at the end of $(0).
- If there is a $LIT directive, MASM places literals at the end of the location counter named on the $LIT directive.

MASM attempts to minimize the space your program requires, by looking first in the list of literals for a value, before allocating a new one. In other words, if you have used the value (+ 27) as a literal

several times, MASM will allocate only one cell with that value, and all of the references to that value will point to the same address. In practice, that means you must never attempt to change the value of a literal, as in this example:

```
        S       A4,(+ 31) .
```

This is pretty obvious; a more subtle example may be when you supply a literal's address as an argument to a subroutine, and the subroutine changes the argument.

Note that you can combine a partial-word modifier of U (that is, an immediate operand) with a literal:

```
        L,U     A4,(' THIS IS A TEST STRING') .
```

MASM stores the string as a literal, and puts the address of the string in the u field of the load instruction. At execution time, the load will put the address of the string in A4.

If there is any question in your mind about what MASM generated for the literal, don't be shy: look for the generated u field in your assembly listing, then find that location in the list of literals at the end of the listing. You might find that the Fieldata constant you expected is really ASCII, or another problem occurred.

9.8 Bitwise Fields: The $FORM Directive

You may recall that MASM can generate a data word divided into equal parts using the '+' operator. But if you need a cell divided on arbitrary bit boundaries, you'll have to use something else: the FORM directive. The general form of a FORM is:

```
label   $FORM   e,e,...,e .
```

The list of expressions 'e' must total 36 or 72; the expressions tell MASM how many bits to put in each field. The name 'label' becomes something like a new directive. You can use 'label' in the operand field of a code line. For example, this $FORM generates the parameters for the Executive Request PRINT$:

```
PF      $FORM   12,6,18 .
```

PF divides a list of three values into fields, respectively 12, 6 and 18 bits. (Giving it more or less than three fields would cause an E flag.) Here is how to use PF:

```
PPARAM  PF      077,L$LEN,L$BUFF .
```

(What these fields mean to PRINT$ is covered later.) The value 077 is placed in a 12-bit field, L$LEN in 6 bits and L$BUFF in 18 bits.

One naming convention for forms uses the bit widths in the name. For example, the form for 12, 6 and 18 bits can be called F12618, or for 16 and 20 bits F1620.

Chapter 3 discussed the 1100 instruction word format. Here is a $FORM to generate an instruction word:

```
I$          $FORM       6,4,4,4,1,1,16          . f,j,a,x,h,i,u
```

If you don't remember each of the fields, you may want to review Chapter 3.

Exercises

E9-1. The table LIST01 contains 2-word entries. Code a sequence to count the entries in which H1 of the second word of each entry is greater than the constant RUNSEQ. (Hint: code a $EQUF for the test field.) Your algorithm must terminate after looking at the last entry. Make any assumptions you need to identify the last entry.

E9-2. Using code from the previous exercise, add the lines needed to save the largest value found in H2 of the first word of each entry. Again, create a $EQUF for the test field.

E9-3. Create another version of your code. This time, you must save not only the highest value found in H2 of the first word of each entry, but the index to that entry. The count requested in E9-1 must continue to work. (Hint: more jumps may be needed to make this work.)

E9-4. Code a $FORM directive called F7920 which will divide a data word into fields of 7, 9 and 20 bits, respectively.

Chapter 10: Looping Instructions and Techniques

10.1 Why Loop?

No operation is so connected with the concept of a stored-program computer as a loop. It is the ability of a computer to repeat a sequence of steps over and over, tirelessly evaluating data and acting on it, which gives computers their power.

We can write sections of code which can execute repetitively, based on a preset counter, or can stop when a required condition has been reached. Each execution of, or "pass", through the loop can typically look at one table entry and branch within the loop based on the contents of an entry.

10.2 How to Code a Counted Loop

Most loops are executed a fixed number of times. That is, the number of times to execute the loop is determined before the loop begins. The loop needs two basic elements to execute the correct number of times:

- An instruction to increment or decrement a loop counter, often a register.
- An instruction to test the value of the loop counter and take action based on the result of the test.

In some cases, it is possible to combine these operations into one instruction. Or, as in Chapter 9's examples, the loop counter and a table pointer (index register) may be the same register. Of course, if you can conveniently use the same register for multiple purposes, the program will require fewer lines of code and take fewer machine cycles to execute, a primary goal of MASM programming.

This chapter covers three main techniques for loop control:

- Add and test,
- JGD, and
- INC, DEC

10.2.1 Add and Test

The final version of the example in Chapter 9 which counts nonzero entries in the SCORES tables shows this technique. It is repeated for your convenience:

```
FIELDA      $EQUF       0,,H1.
FIELDB      $EQUF       0,,H2.
FIELDC      $EQUF       1,,W .
SCRLEN      $EQU        2
SCORES      .
            +           120567,100435 . TWO FIELDS
            +           477522      . SINGLE FIELD
            +           50,200123   . TWO FIELDS
            +           35204       . SINGLE FIELD
            +           12489,78905 . TWO FIELDS
            +           1356078     . SINGLE FIELD
NSCORE      EQU         ($-SCORES)/SCRLEN . NBR OF ENTRIES
            SZ          A0          . INITIALIZE COUNTER
            SZ          A1          . INITIALIZE COUNTER
            SZ          A2          . INITIALIZE COUNTER
            L,U         A3,SCORES   . INITIALIZE INDEX
            LXI,U       A3,SCRLEN   . SET INCREMENT (TABLE SIZE)
LOOP1.
            TZ          FIELDA,A3   . FIELDA NONZERO?
            A,U         A0,1        .  YES,BUMP  NO,
            TZ          FIELDB,A3   . FIELDB NONZERO?
            A,U         A1,1        .  YES,BUMP  NO,
            TZ          FIELDC,*A3  . FIELDC NONZERO?
            A,U         A2,1        .  YES,BUMP  NO,
            L,U         A4,0,A3     . ISOLATE INDEX PORTION
            TLE,U       A4,SCRLEN*NSCORE+SCORES . MAX?
            J           LOOP1       .  NO,LOOP1  YES,
            .
```

The instruction testing FIELDC,*A3 adds the entry size to the index register; the final TLE,U tests the current value of the index register. If the index has reached the maximum, the jump is skipped and the loop is ended.

The last five instructions (beginning with the test of FIELDC) could be replaced by

```
TZ          FIELDC,A3   .
A,U         A2,1        .
TLEM,U      A3,SCRLEN*NSCORE+SCORES .
J           LOOP1       .
```

This removes the L,U A4,0,A3, and removes the * on the A3 in the first of these instructions. TLEM stands for Test Less than or Equal to Modifier. In other words, it does just the test we need, looking only at the modifier portion of X_x. TLEM skips NI if the right half of the contents of U is less than or equal to the modifier portion of X_a. Every time it is executed, after the test is performed, but regardless of the outcome of the test, it bumps the modifier portion of X_a by the contents of the increment portion. (Check your instruction definition table for the formal description.)

Note that TLEM operates on X_a, not X_x, and the incrementation takes place independent of the setting of the h bit. The h bit still controls incrementation of X_x.

10.2.2 Jump Greater and Decrement

If there ever was a "no-brainer" instruction, JGD is it. It works on any register, and it is easy to set up and use. You should review the formal description of the instruction, but here's how to use it:

```
            L           A0,ENTRIES      .
            AN,U        A0,1            .
            S           A0,R5           .
LOOPST      .
            (work of the loop)
            JGD         R5,LOOPST
            (end of the loop)
```

That's all there is to it. Subtract 1 from the number of times you want the loop to be done, put the result in any convenient register and JGD at the end of the loop. Since R registers cannot do useful arithmetic (such as subtracting 1 from the entry count in this case), they tend to get used a lot with JGD. That way, also, the X and A registers are freed up to do what they do best.

JGD uses the j and a fields combined to identify a register; that's how it works on any register. If the named register is greater than zero, the instruction jumps to U; otherwise, it falls through. Regardless of the result of the test, JGD subtracts 1 from the named register each time it is executed.

To see how the operation of the instruction results in one more iteration than the initial value of the register, assume the loop starts with a 1 in R5. The first time JGD is executed, it sees a value greater than zero in R5 and jumps. It also subtracts 1 from R5, making it zero. The next time through, R5 is not greater than zero. The test fails and execution of the loop is ended. If in our example ENTRIES had contained 2, subtracting 1 from 2 leaves 1, and the loop is executed twice.

If you don't want to use an A register to prepare a counter for a JGD, you can do it in the R register, like this:

```
            L           R10,ENTRIES     .
            JGD         R10,$+1         .
```

These 2 instructions replace the first 3 in the last example.

You can use an A or X register with JGD, too. You can even step backward through a table using an X register, if each entry is only one word long.

10.2.3 INC/DEC Loops

INC, DEC, INC2, and DEC2 also combine arithmetic and test functions. If the initial operand or result of the increment/decrement is zero, the instructions execute NI. If nonzero, they skip NI. These instructions go JGD one better: they can work on memory locations as well as registers.

Note that using a memory location as a loop counter is a tiny bit slower than using a register. It takes a little longer to access a memory location than a register.

Here is a sample loop using DEC to maintain and test a loop counter:

```
        L       A5,ENTRIES .
TAG1    .
        (work of the loop)
        DEC     A5          .
        J       OUT         .
        J       TAG1        .
OUT     .
        (end of the loop)
```

This sample decrements the initial count down to zero, just as does the JGD example. The difference is that DEC executes NI when the operand *or result* is zero. You may want to make out a small table to track the value of A5 on each pass (as did the author) to convince yourself. Your table should look like this, if ENTRIES starts out as 3:

Pass	A5	New A5
1	3	2
2	2	1
3	1	0 (ends)

There are separate columns for both A5 and "new A5", because DEC tests both the starting value and the decremented value for zero.

You may notice that the DEC is followed by two jumps in this example, unlike a test instruction, usually followed by one jump. The reason is that DEC executes NI when the operand reaches zero, normally the end of the iterations. By contrast, we usually pick a test instruction which "falls through" (skips NI to something other than a jump instruction) at the end of the iterations. A loop using DEC does not need to adjust the counter the way a JGD counter must be (requiring an A or X register), and your choice of loop counters is more flexible.

You can change the DEC example to use INC like this:

```
        LNA     A5,ENTRIES .
TAG1    .
        (work of the loop)
        INC     A5          .
        J       OUT         .
        J       TAG1        .
OUT     .
        (end of the loop)
```

The value of A5 on each pass is exactly the same as in the DEC example, except that the signs are reversed.

You can use INC2 and DEC2 in a manner similar to INC and DEC. The main difference is that the loop counter should start off with twice the number of times you want the loop to execute. That would normally mean shifting it one bit left to multiply it by two (and extra work). INC2 and DEC2 are not exactly universal tools, but they are available for special cases.

10.3 Loops Without Counters

Many times, you need to step through a list, searching for an entry which satisfies certain criteria. The code doesn't need to know how many entries there are, or the end of the list is a specially-marked entry. Here is the pattern:

```
LOOPST      .
            (work of the loop)
            TNZ         LINK,X6     . END OF LIST?
            J           OUT         .  YES,OUT  NO,
            (more work of the loop)
            J           LOOPST      . DO IT AGAIN
OUT         .
            (end of the loop)
```

The unconditional jump begins the loop over again. A loop of this sort must be coded carefully to make sure which it will terminate.

Exercises

E10-1. Code a loop which will quit after it has been executed GRAND1 times. Use the add and test technique.

E10-2. Recode the loop in the previous exercise using the JGD technique.